DESIGN PATTERNS - CHAPITRE 9

Composite en Python

Traiter des feuilles et des groupes avec une opération commune et une agrégation récursive.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Traiter des feuilles et des groupes avec une opération commune et une agrégation récursive.
- Distinguer le contrat commun des opérations réservées aux branches.

Prérequis. Savoir utiliser classes, héritage, composition, listes, dictionnaires et exceptions Python. Les mécanismes spécifiques sont expliqués avant leur emploi.

Composite : intention et expérience

Composite : traiter une feuille et un groupe

Un **Composite** partage une opération entre objets élémentaires et groupes. Le groupe délègue à ses enfants, qui peuvent aussi être des groupes. Dans cet arbre de dessins, le client appelle `render()` sans connaître la profondeur.

Python

```python
# example.py
# Component
class Graphic:
    def render(self):
        pass

# Leaf
class CircleGraphic(Graphic):
    def __init__(self, radius):
        self.radius = radius

    def render(self):
        return f"Rendering circle with radius {self.radius}"

# Leaf
class SquareGraphic(Graphic):
    def __init__(self, side):
        self.side = side

    def render(self):
        return f"Rendering square with side {self.side}"

# Composite
class CompoundGraphic(Graphic):
    def __init__(self):
        self.children = []

    def add(self, graphic):
        self.children.append(graphic)

    def remove(self, graphic):
        self.children.remove(graphic)

    def render(self):
        result = ""
        for child in self.children:
            result += child.render() + "\n"
```

```
        return result

# Client code
if __name__ == "__main__":
    compound_graphic = CompoundGraphic()
    compound_graphic.add(CircleGraphic(5))
    compound_graphic.add(SquareGraphic(10))

    compound_graphic2 = CompoundGraphic()
    compound_graphic2.add(CircleGraphic(2))
    compound_graphic.add(compound_graphic2)

    print(compound_graphic.render())
```

Composite : définir le contrat commun

Le contrat commun utile à toutes les feuilles et branches est ici `operation() -> str`. Seules les branches acceptent des enfants. Les méthodes `add` et `remove` héritées par une feuille lèvent donc `NotImplementedError` ; une autre conception pourrait les exposer uniquement sur les branches pour éviter ces opérations invalides.

Cette classe de base est abstraite par convention : sans `ABC`, Python n'interdit pas son instanciation. L'arbre ne doit pas contenir de cycle, sinon la récursion ne termine pas.

Python

```python
# example.py
class Component:
    """
    Declare the interface for objects in the composition.
    This includes methods for adding, removing, and getting child components,
    as well as the 'operation' to be carried out.
    """
    def __init__(self, name):
        self._name = name

    def operation(self):
        """
        The 'Operation' that all components carry out.  Subclasses must
implement this.
        """
        raise NotImplementedError("Subclasses must implement operation()")

    def add(self, component):
        """
        Add a child component.  Leaf nodes won't implement this.
        """
        raise NotImplementedError("Subclasses must implement add()")

    def remove(self, component):
        """
        Remove a child component. Leaf nodes won't implement this.
        """
        raise NotImplementedError("Subclasses must implement remove()")

    def get_child(self, i):
        """
        Get a child component. Leaf nodes won't implement this.
        """
        raise NotImplementedError("Subclasses must implement get_child()")


class Leaf(Component):
```

```
    """
    Represent the leaf objects in the composition. A leaf has no children and
implements the operation directly.
    """
    def operation(self):
        """
        Leaf-specific implementation of the operation.
        """
        return f"Leaf {self._name}: Doing leaf operation."


class Composite(Component):
    """
    Represent a composite object. A composite can have children, which can be
either Leafs or other Composites.
    """
    def __init__(self, name):
        super().__init__(name)
        self._children = []

    def add(self, component):
        """
        Adds a child component to the composite.
        """
        self._children.append(component)

    def remove(self, component):
        """
        Removes a child component from the composite.
        """
        self._children.remove(component)

    def get_child(self, i):
        """
        Returns the child component at the specified index.
        """
        return self._children[i]

    def operation(self):
```

```
        """
        Executes the operation on all child components and aggregates their
results.
        """
        results = []
        for child in self._children:
            results.append(child.operation())
        return f"Branch {self._name}: " + ", ".join(results)
```

Suite des classes du bloc précédent : construisez une racine, deux branches et trois feuilles. Prévoyez le résultat avant l'exécution, puis supprimez une feuille. L'agrégation est récursive : chaque branche combine les résultats de ses enfants.

Python

```python
# example.py
# Client code
if __name__ == "__main__":
    # Create a tree structure
    root = Composite("Root")
    branch1 = Composite("Branch1")
    branch2 = Composite("Branch2")

    leaf1 = Leaf("Leaf1")
    leaf2 = Leaf("Leaf2")
    leaf3 = Leaf("Leaf3")

    root.add(branch1)
    root.add(branch2)

    branch1.add(leaf1)
    branch1.add(leaf2)

    branch2.add(leaf3)

    # Execute 'operation' on the root (composite)
    print(root.operation())
    # Expected Output: Branch Root: Branch Branch1: Leaf Leaf1: Doing leaf operation., Leaf Leaf2: Doing leaf operation., Branch Branch2: Leaf Leaf3: Doing leaf operation.

    # Execute 'operation' on a leaf
    print(leaf1.operation())
    # Expected Output: Leaf Leaf1: Doing leaf operation.

    #Demonstration of removing a component
    branch1.remove(leaf1)
    print(root.operation())
    # Expected Output: Branch Root: Branch Branch1: Leaf Leaf2: Doing leaf operation., Branch Branch2: Leaf Leaf3: Doing leaf operation.
```

Vérifier et interpréter

Retirez une feuille d'une branche puis recalculez l'opération de la racine. Seule la contribution de cette feuille doit disparaître. Une structure arborescente simplifie l'agrégation ; elle impose aussi de décider qui possède les enfants, si un enfant peut avoir plusieurs parents et comment interdire les cycles.

Version web : https://www.clearcode.fr/courses/programming/design-patterns/structure/composite